BUSINESS INTELLIGENCE AND DATA ANALYTICS

University of Petra

Institute of Analytics Recognized Programme

University of Petra

Introduction to Neural Networks

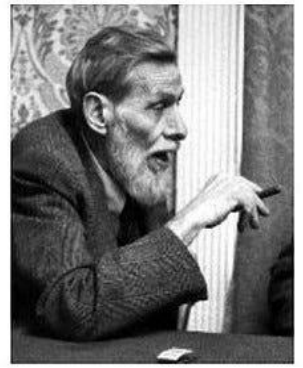
307307 BI Methods

Topics

- Fundamentals of neural networks
- Evolution from single Perceptrons to MLPs
- Detailed MLP architecture (input, hidden, and output layers)
- Mathematical representations
- Various activation functions (Sigmoid, ReLU, etc.)
- Backpropagation and training methodologies
- Loss functions and optimization techniques
- Architecture design considerations
- Real-world applications
- Advantages and limitations
- Modern MLP variants and implementations

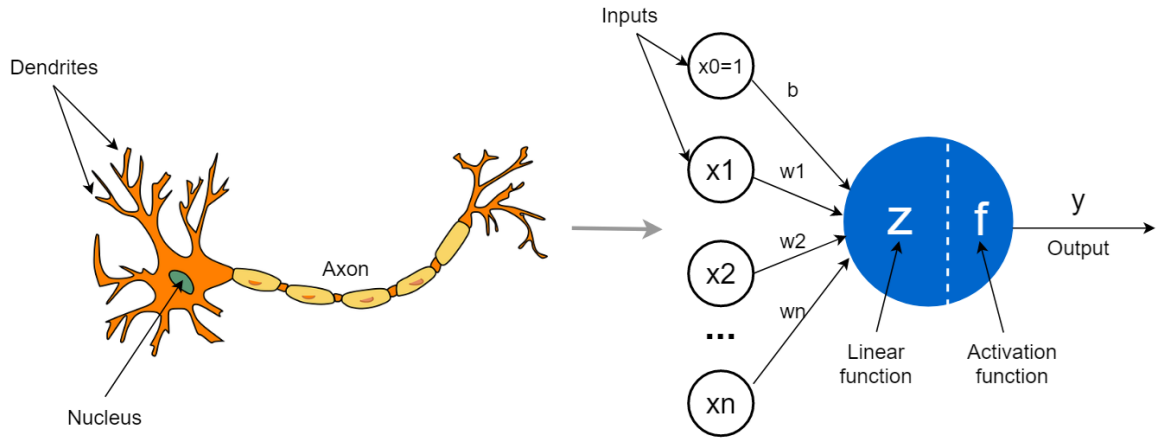
What is a Neural Network?

- To understand how LLMs work, it helps to understand some fundamentals of neural networks, as they are the foundation of many AI systems.
- Neural networks are inspired by the brain and consist of interconnected “neurons” that adjust connection strengths during learning.
- 1943 – McCulloch & Pitts: Proposed the first mathematical model of a neuron (MCP neuron), a binary threshold logic gate.
- They showed that networks of such neurons could compute any logical function.
- This laid the theoretical foundation for artificial neural networks.



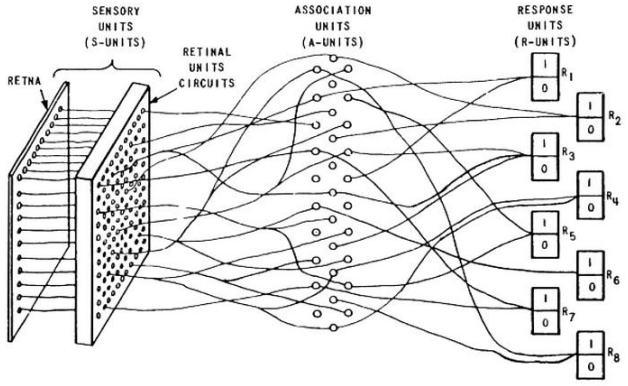
Warren Sturgis McCulloch
(1898 – 1969)

Walter Harry Pitts, Jr.
(1923 – 1969)

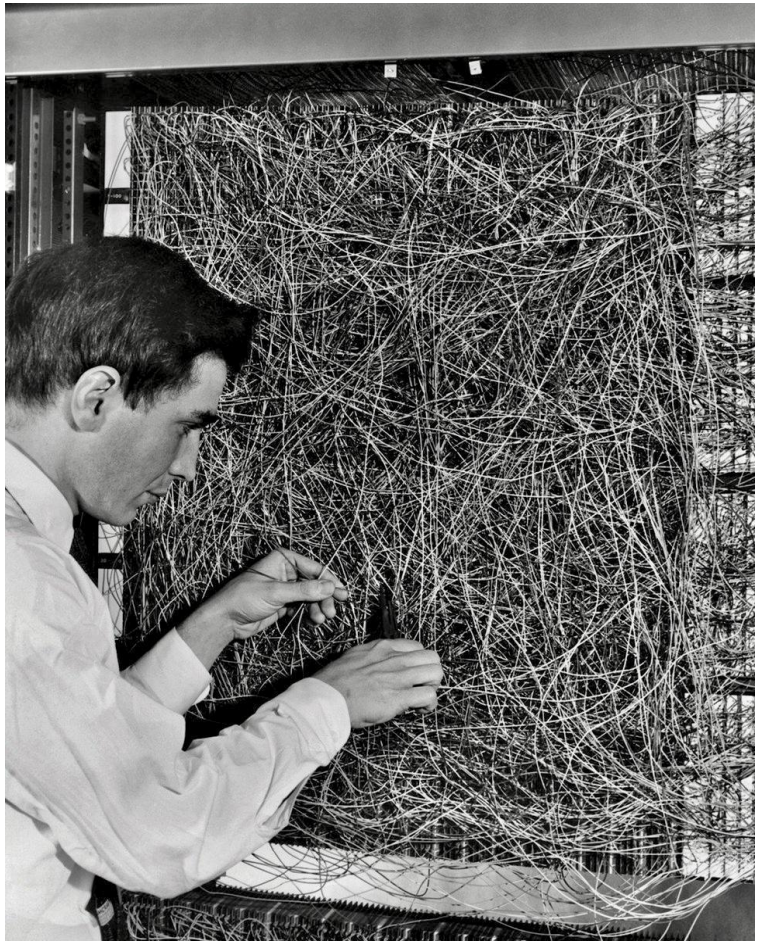


The Perceptron: Building Block of Neural Networks

- 1958 – Frank Rosenblatt (inspired by McCulloch & Pitts): Introduced the **Perceptron** (the Mark I Perceptron at Cornell).
- Designed to recognize patterns in visual data.
- It was Implemented in hardware.
- The perceptron is a binary classifier: input → weighted sum → threshold → output.
- Rosenblatt also described multi-layer perceptrons, but at the time no algorithm existed to train them effectively.



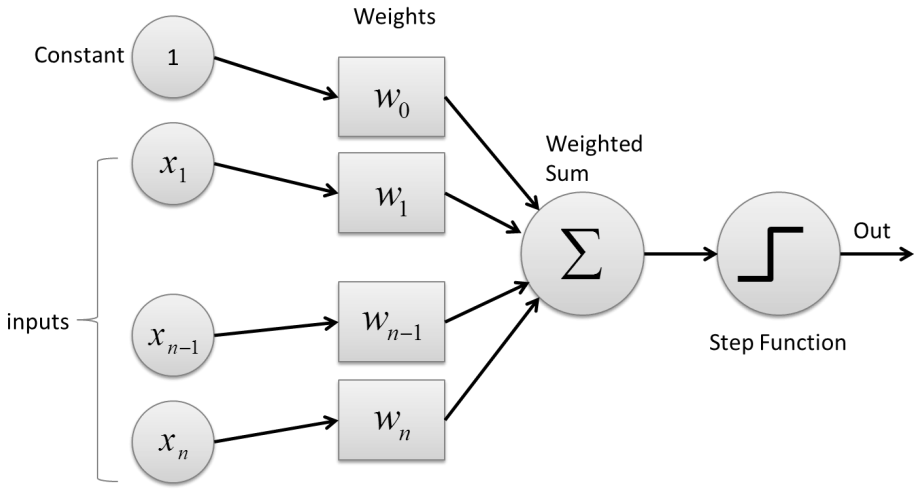
F. Rosenblatt



The diagram shows **Rosenblatt's Perceptron (1958)**, which is essentially an early form of a **multi-layer perceptron (MLP)**: sensory units (S-units) connected to association units (A-units), which in turn connect to response units (R-units). Inspired by the visual cortex, it could learn to classify input patterns by adjusting connection weights. While the theory was purely mathematical, Rosenblatt also built an **electromechanical implementation called the Mark I Perceptron**, which used an array of photocells as the retina, analog circuits for weighted connections, and motors to adjust the weights physically. This made it one of the first tangible demonstrations of machine learning hardware.

The Perceptron

- Inputs: $x_1, x_2, \ldots, x_n$
- Weights: $w_1, w_2, \ldots, w_n$
- Bias: b
- Activation function: Step function
- Output: 1 if weighted sum > threshold, 0 otherwise

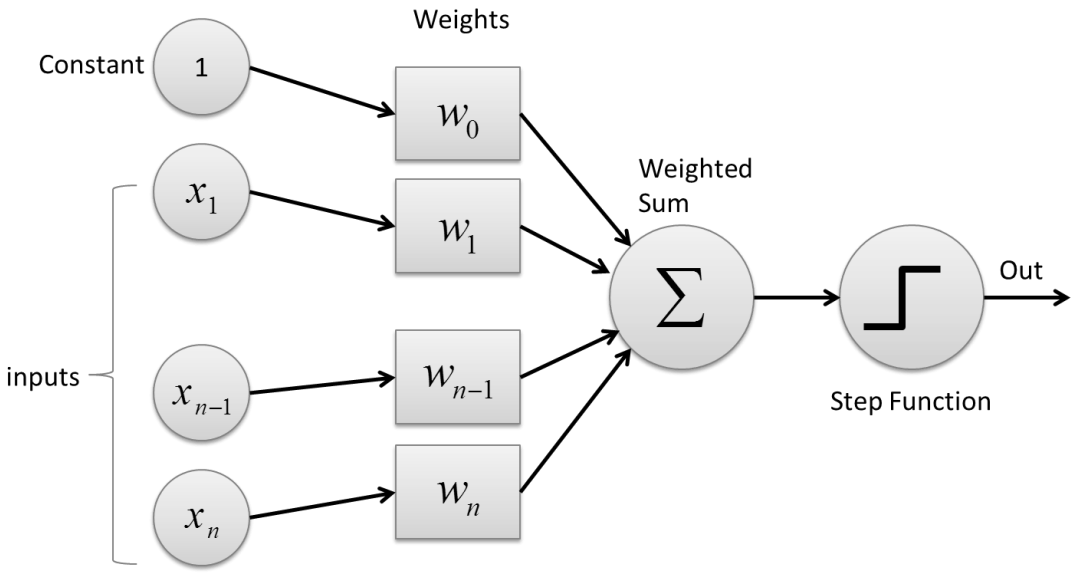


How a Perceptron Works

1. Multiply each input by its corresponding weight
2. Sum all weighted inputs
3. Add the bias term
4. Apply the activation function
5. Output the result

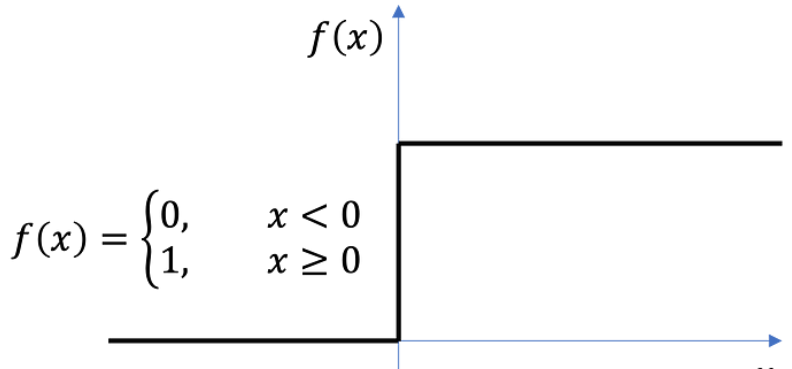
Mathematically:

- $z = w_1x_1 + w_2x_2 + ... + w_nx_n + b$
- output = activation(z)



The Step Function:

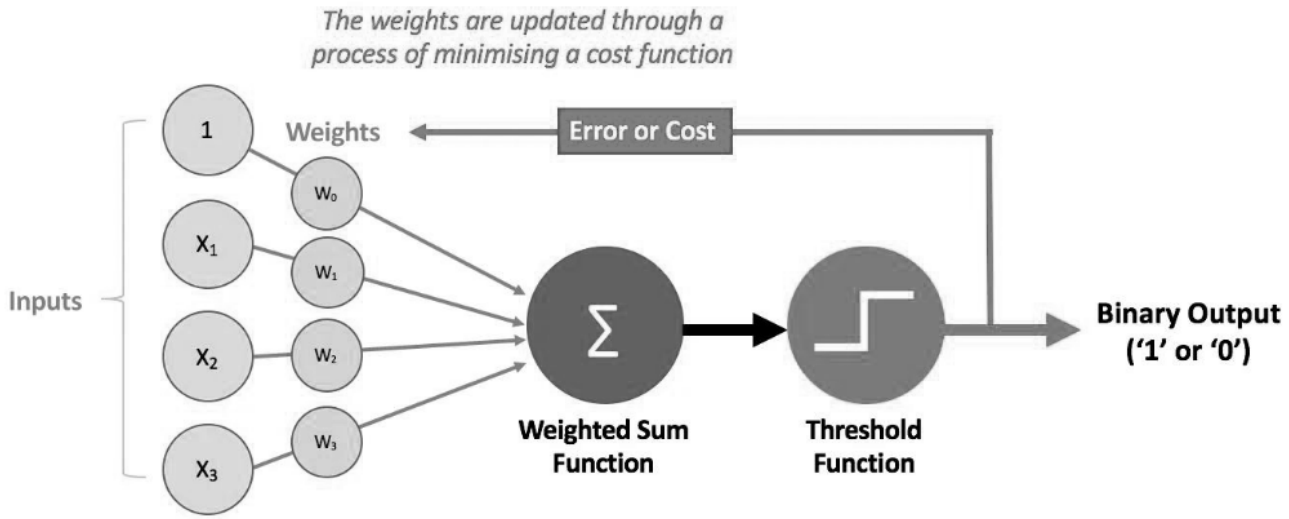
- Used in original Perceptron
- Output: 1 if z ≥ 0, 0 if z < 0
- Not differentiable at 0



How Perceptron Learn

For each training example:

1. Calculate predicted output y_pred
2. Calculate error: error = y_true - y_pred
3. Update weights: w_new = w_old + learning_rate * error * x
4. Update bias: b_new = b_old + learning_rate * error



Step-by-Step Hand Calculation for AND Gate

Let's work through the perceptron learning algorithm by hand for the AND gate:

- Training data: X = [[0,0], [0,1], [1,0], [1,1]], y = [0, 0, 0, 1]
- Learning rate (η) = 0.1
- Initial weights (randomly assigned): $w_1 = 0.3$, $w_2 = -0.1$
- Initial bias: b = 0.2

First Iteration Starting Next Page

Training data

A (Input 1)	B (Input 2)	X = (A.B)
0	0	0
0	1	0
1	0	0
1	1	1

Independent Variables

Dependent Variable

Step-by-Step Hand Calculation for AND Gate

Record #1: (0,0) → 0

- Inputs: $x_1 = 0, x_2 = 0$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(0) + 0.2 = 0.2$
- Activation: output = 1 (since $z > 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * error * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
  - $b = b + \eta * error = 0.2 + 0.1 * (-1) = 0.1$

Record #2: Inputs: $x_1 = 0$, $x_2 = 1$, output = 0

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(1) + 0.1 = 0$
- Activation: output = 1 (since $z \geq 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1

Weight updates:

- $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
- $w_2 = w_2 + \eta * error * x_2 = -0.1 + 0.1 * (-1) * 1 = -0.2$
- $b = b + \eta * error = 0.1 + 0.1 * (-1) = 0$

Step-by-Step Hand Calculation for AND Gate

Record #3: Inputs: $x_1 = 1$, $x_2 = 0$, output = 0

•Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(1) + (-0.2)(0) + 0 = 0.3$

•Activation: output = 1 (since $z > 0$)

•True output: $y = 0$

•Error: error = y - output = 0 - 1 = -1

•Weight updates:

•$w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 1 = 0.2$

•$w_2 = w_2 + \eta * error * x_2 = -0.2 + 0.1 * (-1) * 0 = -0.2$

•$b = b + \eta * error = 0 + 0.1 * (-1) = -0.1$

- **Record #4: Inputs: $x_1 = 1$, $x_2 = 1$, output = 1**
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.2(1) + (-0.2)(1) + (-0.1) = -0.1$
- Activation: output = 0 (since $z < 0$)
- True output: $y = 1$
- Error: error = y - output = 1 - 0 = 1
- Weight updates:
- $w_1 = w_1 + \eta * error * x_1 = 0.2 + 0.1 * 1 * 1 = 0.3$
- $w_2 = w_2 + \eta * error * x_2 = -0.2 + 0.1 * 1 * 1 = -0.1$
- $b = b + \eta * error = -0.1 + 0.1 * 1 = 0$

End of Iteration 1:

- Updated weights: $w_1 = 0.3$, $w_2 = -0.1$
- Updated bias: $b = 0$

Second Iteration

Record # 1: Inputs: $x_1 = 0$, $x_2 = 0$, Output = 0

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(0) + 0 = 0$
- Activation: output = 1 (since $z \geq 0$)
- True output: y = 0
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * error * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
  - $b = b + \eta * error = 0 + 0.1 * (-1) = -0.1$

Example 2: Inputs: $x_1 = 0$, $x_2 = 1$, output = 0

- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(1) + (-0.1) = -0.2$
- Activation: output = 0 (since $z < 0$)
- True output: y = 0
- Error: error = y - output = 0 - 0 = 0
- Weight updates (no change as error = 0):
  - $w_1 = 0.3$
  - $w_2 = -0.1$
  - $b = -0.1$

After several iterations, the perceptron will converge to weights that correctly classify all AND gate examples.

Python Implementation

```python
from sklearn.linear_model import Perceptron
import numpy as np

  # Training data for AND gate
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

  # Initialize and train Perceptron
model = Perceptron(max_iter=100, eta0=0.1,
  random_state=42)
model.fit(X, y)
# Results
print("Weights:", model.coef_)
print("Bias:", model.intercept_)
print("Predictions:", model.predict(X))
```

```
Weights: [[0.2 0.2]]
Bias: [-0.2]
Predictions: [0 0 0 1]
```

The code shows a scikit-learn Perceptron implementation for the AND gate problem.

The code:

1. Imports NumPy, scikit-learn's Perceptron, and matplotlib
2. Sets up the training data for the AND gate
3. Initializes a Perceptron with 100 max iterations and a random seed of 42
4. Trains the perceptron on the AND gate data
5. Prints the learned weights, bias, and predictions

The output shows:

- **Weights: [[0.2 0.2]]** - The perceptron learned to assign a weight of 0.2 to both inputs
- **Bias: [-0.2]** - The bias is -0.2
- **Predictions: [0 0 0 1]** - The perceptron correctly classified all four examples of the AND gate

With these weights and bias, the decision function is: $0.2\times(\text{input1}) + 0.2\times(\text{input2}) - 0.2$

For the four input combinations:

- [0,0]: $0.2\times0 + 0.2\times0 - 0.2 = -0.2 < 0 \rightarrow$ output 0
- [0,1]: $0.2\times0 + 0.2\times1 - 0.2 = 0 \rightarrow$ output 0
- [1,0]: $0.2\times1 + 0.2\times0 - 0.2 = 0 \rightarrow$ output 0
- [1,1]: $0.2\times1 + 0.2\times1 - 0.2 = 0.2 > 0 \rightarrow$ output 1

This perceptron implements the AND gate logic.

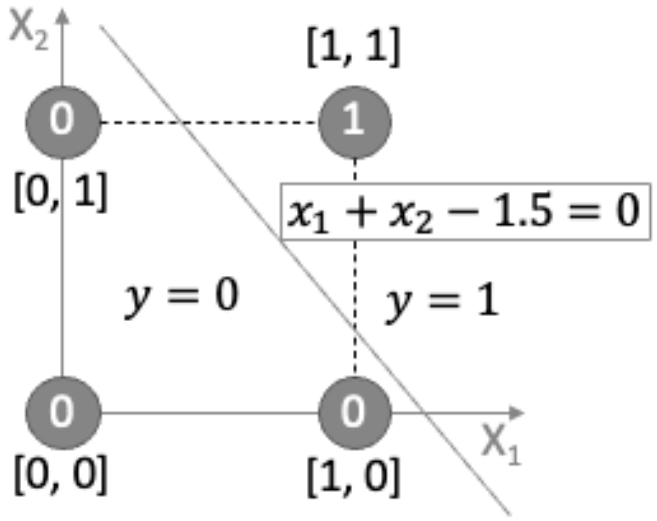
The decision boundary is the line $2x_1 + 2x_2 - 0.2 = 0$, which separates the point (1,1) from the other three points.

Open in Colab

Decision Boundary

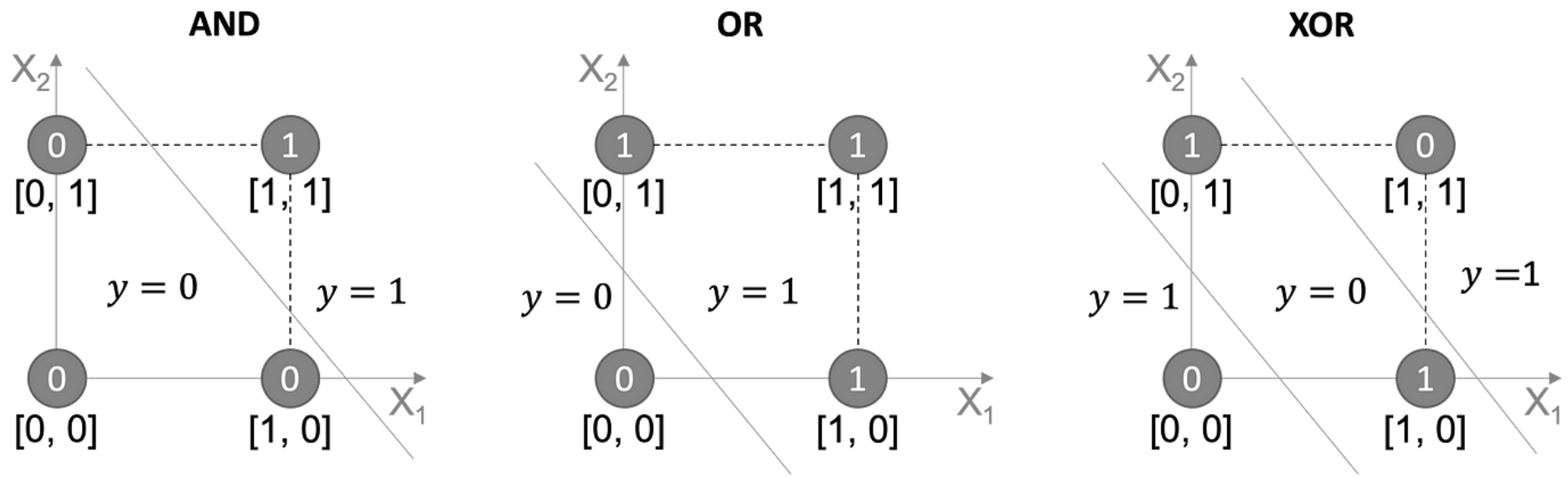
- The perceptron learns a decision boundary: $w_1x_1 + w_2x_2 + b = 0$
- Points above the line are classified as 1
- Points below the line are classified as 0
- For AND gate, only the point (1,1) should be above the line

X_1	X_2	y
0	0	0
0	1	0
1	0	0
1	1	1



Limitations of Single Layer Perceptron

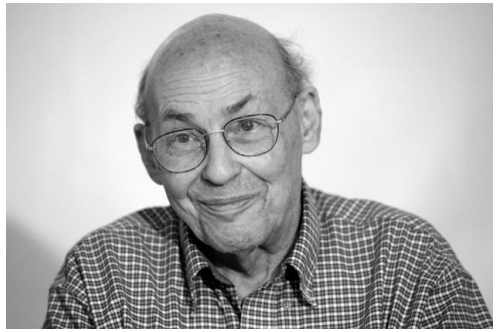
- Single layer perceptron can only learn linearly separable patterns
- It cannot solve XOR problem (need multiple layers)
- Simple update rule isn't suitable for complex problems
- There was no suitable algorithm to train a multi-layer perceptron.



The Multi-Layer Perceptron (MLP)

Limitations of the Perceptron

While useful for linearly separable problems, the single perceptron cannot solve complex problems like XOR classification, as demonstrated by Minsky and Paper in their 1969 book "Perceptrons.“



The Multi-Layer Perceptron

The Multi-Layer Perceptron addresses the limitations of the single perceptron by introducing:

- Multiple layers of neurons
- Non-linear activation functions
- More sophisticated learning algorithms

Structure of an MLP

Definition: An MLP is a class of feedforward artificial neural network that consists of at least three layers of nodes: **input**, **hidden**, and **output** layers.

Key Feature: Each neuron in one layer is connected to every neuron in the next layer (fully connected).

1. Input Layer

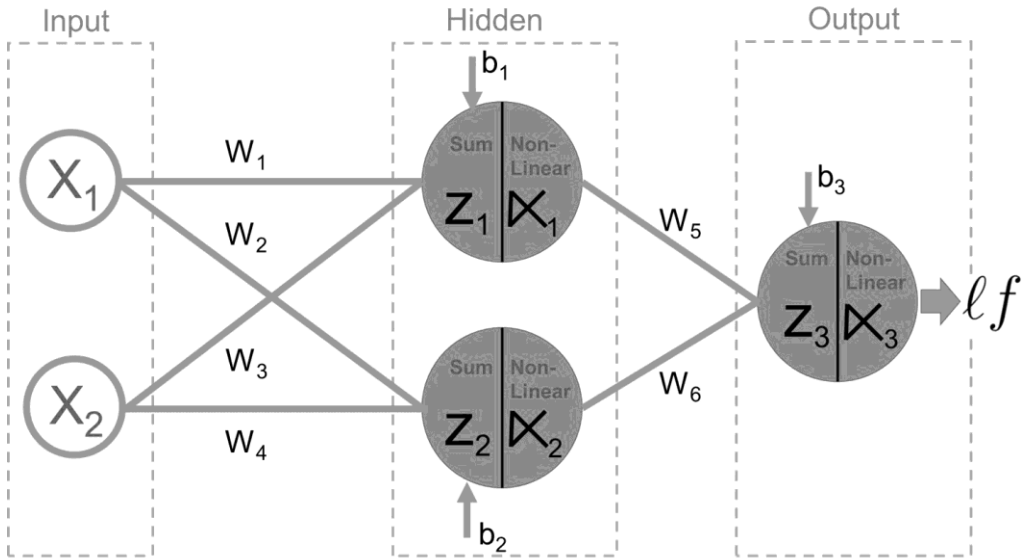
- Receives the raw input features
- One neuron per input feature
- No computation occurs here; inputs are simply passed forward

2. Hidden Layer(s)

- One or more layers between input and output
- Each neuron in a hidden layer:
- Receives inputs from all neurons in the previous layer
- Computes a weighted sum
- Applies a non-linear activation function
- Passes the result to the next layer

3. Output Layer

- Produces the final prediction or classification
- Structure depends on the task:
    - Regression: Often a single neuron with linear activation
    - Binary classification: One neuron with sigmoid activation
    - Multi-class classification: Multiple neurons (one per class) with softmax activation



Structure of an MLP

3. Neurons and Connections

- Each neuron computes a weighted sum of inputs and applies an activation function
- Fully connected between layers (dense connections)

4. Activation Functions

- Introduce non-linearity
- Examples: ReLU, Sigmoid, Tanh, Softmax

5. Loss Function

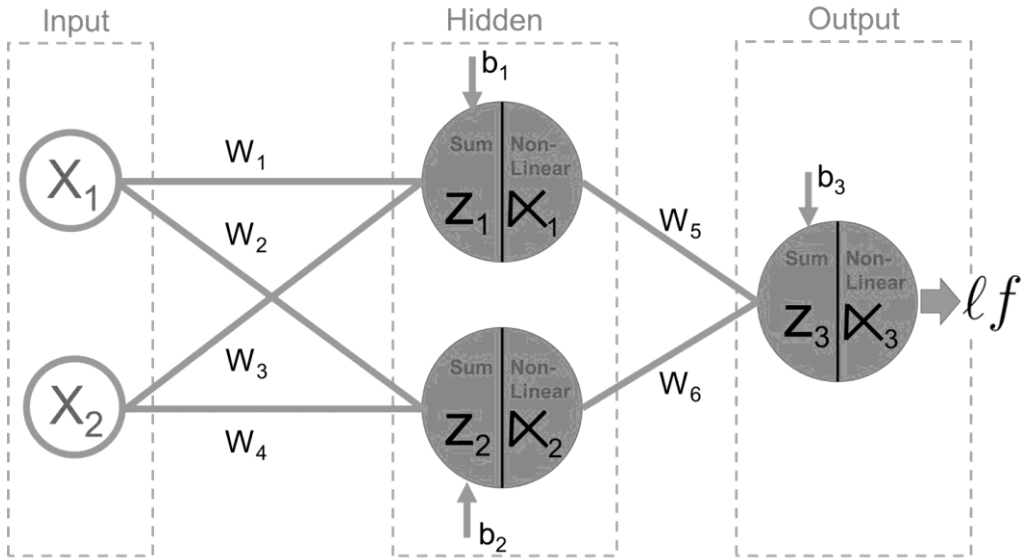
- Measures the error between predicted and true outputs
- Examples: Mean Squared Error, Cross-Entropy

6. Optimizer

- Updates weights to minimize loss
- Examples: SGD, Adam

7. Training Data

- Labeled data used to train the network
- Split into training, validation, and test sets



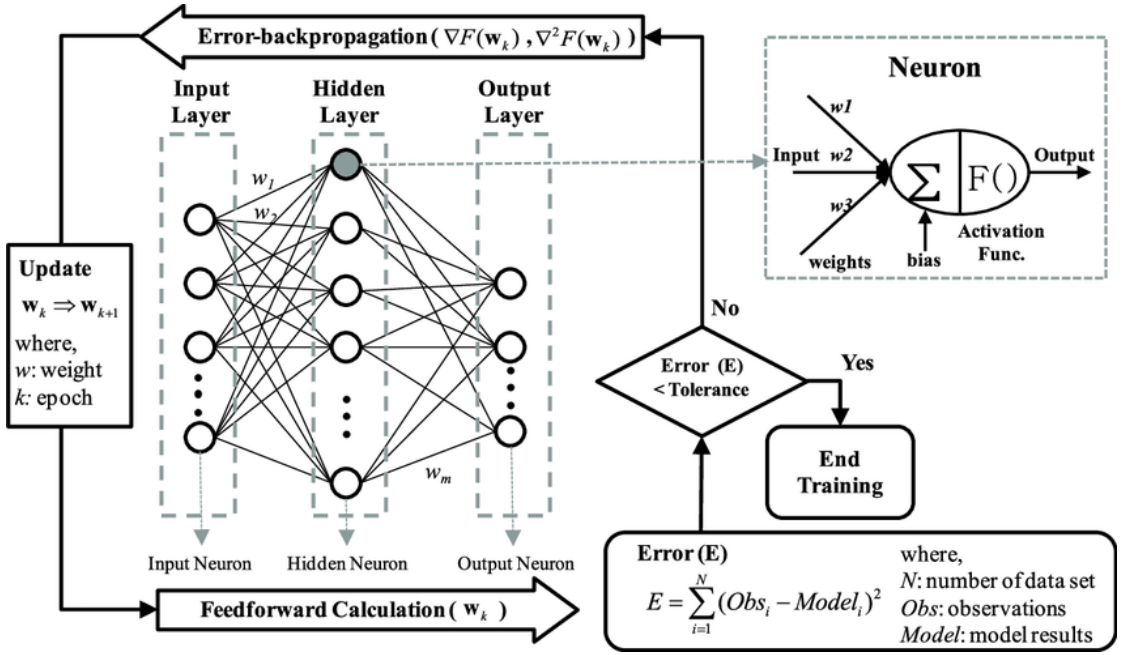
How an MLP Learn – Step by Step

Forward Propagation

- Input features pass through the network layer by layer.
- Each neuron computes a weighted sum of inputs and applies an activation function.
- The final layer produces a prediction.

Loss Computation

- A loss function measures the difference between predicted and actual outputs.
- Common loss functions:
  - Mean Squared Error (regression)
  - Cross-Entropy Loss (classification)



How an MLP Learn – Training Process

Backpropagation

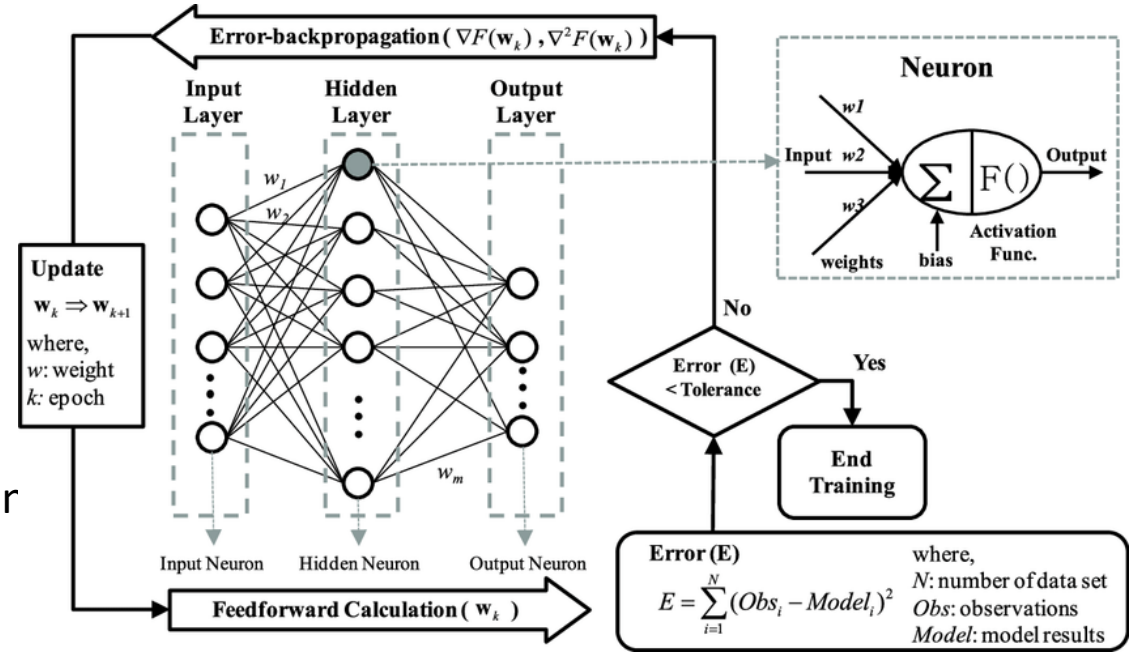
- Gradients of the loss are calculated with respect to each weight using the chain rule.
- This identifies how each weight contributed to the error.

Weight Update

- An optimizer (e.g., SGD) adjusts weights to reduce loss:
  - w := w - learning_rate × gradient
- This process repeats over multiple iterations (epochs) usir training data.

Goal

- Gradually minimize the loss and improve prediction accuracy.
- **Loss Function**: MSE for regression, Cross-Entropy for classification.
- **Optimization**: Backpropagation + Gradient Descent (or Adam).



Activation Functions other than Step Function

- Neural Networks use activation functions other than the simple step function in the Perceptron.
- Activation Function helps the neural network use important information while suppressing irrelevant data points (i.e., allows local “gating” of information).

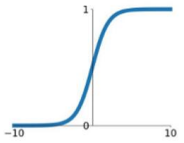
Common Activation Functions: A Toolbox for Neural Networks

- **Sigmoid**: Compresses input to a range between 0 and 1.
- **Tanh**: Like sigmoid but ranges from -1 to 1.
- **ReLU**: Passes positive values, zeroes out negatives.
- **Leaky ReLU**: Allows a small, non-zero gradient for negative inputs.
- **Maxout**: Chooses the most active linear response.
- **ELU**: Smoothly transitions through zero and allows negative outputs.

Each function offers a trade-off between simplicity, flexibility, and computational cost—choose based on the task and depth of your model

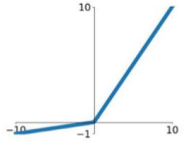
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



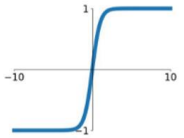
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

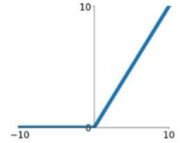


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

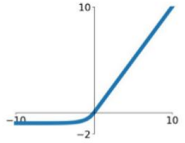
ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



https://ml-explained.com/blog/activation-functions-explained

The Difference Between a Perceptron and an MLP?

Feature	Perceptron	Multilayer Perceptron (MLP)
Layers	Only 1 layer (no hidden layers)	2+ layers (has hidden layers)
Activation Function	Step function (hard threshold)	Nonlinear (e.g., ReLU, sigmoid, tanh)
Learning Rule	Simple rule: update on error	Gradient descent + backpropagation
Tasks It Can Solve	Only linearly separable problems	Nonlinear, complex problems

Train a Neural Network to Classify the MINST Dataset

- The MNIST (Modified National Institute of Standards and Technology database) dataset contains a training set of 60,000 images and a test set of 10,000 images of handwritten digits.
- The handwritten digit images have been size-normalized and centered in a fixed size of 28×28 pixels.
- The MNIST digits dataset is often used by data scientists who want to try machine learning techniques and pattern recognition methods on real-world data while spending minimal effort on preprocessing and formatting.

Train a Neural Network to Classify the MINST Dataset

```
from tensorflow import keras
from tensorflow.keras import layers


# 1. Load example data (MNIST digits)
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # normalize to [0,1]

# 2. Build a simple neural network
model = keras.Sequential([
    layers.Flatten(input_shape=(28, 28)),  # flatten 28x28 images
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# 3. Compile and train
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

# 4. Evaluate
print("Evaluating the Model")
model.evaluate(x_test, y_test)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
/usr/local/lib/python3.12/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning:
  super().__init__(**kwargs)
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.8787 - loss: 0.4332
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9650 - loss: 0.1218
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9756 - loss: 0.0783
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9822 - loss: 0.0579
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9868 - loss: 0.0438
Evaluating the Model
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9721 - loss: 0.0928
[0.0808534249663353, 0.9760000109672546]
```

Open in Colab